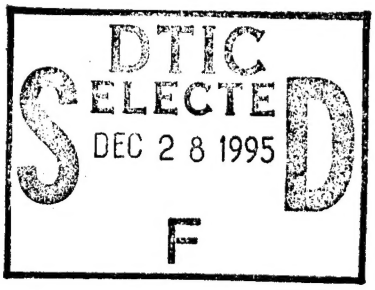
NASA Contractor Report 198232

ICASE Report No. 95-74

ICASE

RUNTIME VOLUME VISUALIZATION FOR PARALLEL CFD



Kwan-Liu Ma

NASA Contract No. NAS1-19480
October 1995

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23681-0001

Operated by Universities Space Research Association

19951222 015

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

Runtime Volume Visualization for Parallel CFD

Kwan-Liu Ma[†]
Institute for Computer Applications in Science and Engineering
Mail Stop 132C
NASA Langley Research Center
Hampton, Virginia 23681-0001

Accesion For
NTIS CRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification .....
By .....
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Abstract

This paper discusses some aspects of the design of a data distributed, massively parallel volume rendering library for runtime visualization of parallel computational fluid dynamics simulations in a message-passing environment. Unlike the traditional scheme in which visualization is a postprocessing step, the rendering is done in place on each node processor. Computational scientists who run large-scale simulations on a massively parallel computer can thus perform interactive monitoring of their simulations. The current library provides an interface to handle volume data on rectilinear grids. The same design principles can be generalized to handle other types of grids. For demonstration, we run a parallel Navier-Stokes solver making use of this rendering library on the Intel Paragon XP/S. The interactive visual response achieved is found to be very useful. Performance studies show that the parallel rendering process is scalable with the size of the simulation as well as with the parallel computer.

[†]This research was supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction

For many scientific and engineering problems, computational fluid dynamics (CFD) has become increasingly popular as a means to gather information for design and analysis purposes. Massively parallel computing offers both the computing power and memory space required to attain the desired accuracy and turnaround time for CFD calculations. Traditionally, monitoring of typical parallel CFD calculations is done by transmitting, at regular times, a subset of the solutions back to a host computer. If the user just wants to track a few numbers at some particular spatial positions, then only a small amount of data is transferred. However, if the user needs to analyze or *visualize* the overall flowfield, the amount of data transferred could be enormous, possibly ranging from several megabytes to gigabytes. Acquiring the whole distributed domain of data involves expensive operations. Storing and postprocessing the data on another computer could also be problematic.

A better approach is to analyze data in place on each processor at the time of the simulation. The locally analyzed results, which are usually in a more economical form, e.g. a two-dimensional image, are then combined and sent back to a host computer for viewing. Scientific visualization is an effective data analysis method making use of computer graphics techniques, and volume rendering has been recognized as one of the most direct ways for visualizing three-dimensional data. This paper describes the design of a parallel volume rendering (PVR) library which renders in-place distributed data on a rectilinear grid. There are special considerations for such design and implementation. In this paper, our discussion focuses on the PVR algorithm. In [1], a thorough discussion is given for the library design of a parallel *polygon* renderer.

We performed tests for runtime visualization of a three-dimensional Navier-Stokes solver on the Intel Paragon XP/S using from eight to 216 processors. The interactive visual response achieved is found to be very helpful. The realistic pictures of the overall or partial flowfield help not only monitor but also understand the simulation. Performance studies show that the parallel rendering process is scalable with the size of the simulation as well as with the parallel computer. Although our current implementation only handles data on rectilinear grids, the em design principles of the library can be generalized to handle unstructured or curvilinear grids as well. A PVR algorithm developed for unstructured-grid data is described in another paper [3].

2 Volume Visualization

Direct volume rendering is a powerful visualization technique. It can render the flowfield realistically as a semi-transparent gel or smoke-like cloud. However, direct volume rendering involves very computationally intensive calculations and slow rendering rates has limited its

use. While massively parallel computers are becoming more accessible, interactive rendering of large data sets now can be achieved by using, for example, a 32-node intel Paragon [9] or the more powerful IBM SP2.

There are two approaches for direct volume rendering: projection and ray-casting. For this work, ray-casting is used because it is easier to parallelize and implement; furthermore, it can also render cut-planes and iso-surfaces. In the ray-casting approach, an image is constructed in image order by casting rays from the eye, through the image plane and into the data volume. One ray per pixel is generally sufficient, provided that the image sample density is higher than the volume data resolution. The data volume is sampled along the ray, usually at a rate of one to two samples per voxel (volume element). At each sample, a color and opacity is computed by interpolating from the data values. For a parallelepiped element, there are eight vertices and trilinear interpolation is often used. The final image value corresponding to each ray is formed by compositing, front-to-back, the color as well as the opacity of the samples along the ray. The color/opacity compositing operation is *associative*, which allows us to break a ray up into segments, process the sampling and compositing of each segment independently, and combine the results from each segment via a final merging step. This is the basis for our PVR algorithm [4].

3 Design Considerations for a PVR Library

The design and implementation of a software library involves several key issues such as the application programmer interface (API), portability, performance and versatility. The library should have a simple interface that does not intimidate the user. A good library should mask differences across system software and hardware platforms through abstraction. Good performance is essential so the user will not be tempted to write custom code. Ideally, a library should improve performance beyond what an average user could easily achieve without it. Finally the library should adapt to unforeseen situations, such as low memory restrictions, and to more sophisticated application problems. Based on the above principles, design considerations essential to implementing a PVR library include:

- Parallelizing direct volume rendering:
  - Data distribution
  - Load balancing
  - Memory constraints
  - Scalability
  - Parallel resampling
  - Parallel compositing

- Portability
- Image output and display
- Library interface

Like most other parallel applications, parallelizing direct volume rendering is a divide and conquer process. The goal is to distribute both data and computation among available processors. There are basically two approaches for parallelizing direct volume rendering: one is to separate the ray-casting (i.e. resampling) process from the compositing process [4]; the other is to overlap them [9, 3]. The first approach has been selected for rendering regular data because it is straightforward to implement and can render as efficiently as the overlapping approach in a runtime visualization setting.

Data distribution is a problem for both the application (parallel CFD) and the visualization (PVR). The design principle is to allow the application programmer to focus on the application program instead of the rendering program. Ideally, a renderer should be able to process local data regardless of how decomposition was done. For both parallel CFD and PVR, it has been shown the best scalability can be achieved by decomposing the domain in such a way as to have sizes in all dimensions as close as possible [10, 7]. For example, three-dimensional decomposing (into blocks) is better than one-dimensional (into slabs); for parallel CFD, block decomposition could reduce communication costs, and for PVR, rendering would become less view-dependent. Currently, the renderer can handle one-, two-, and three-dimensional decomposed data, but each data subset must contain voxels that are spatially consecutive; this restriction can be relaxed for irregular data partitioning [3]

Load balancing is an important issue for achieving the maximum parallel efficiency. According to our test results, imbalanced load generally can degrade the overall performance by 20-40%. Dynamic load balancing, if implemented efficiently, should be able to remove at least 50% of the degradation. A few load balancing strategies for PVR have been proposed which require preprocessing of the data, or specialized parallel architectures or data distribution schemes [6, 9, 8].

For time-varying data, preprocessing is generally impractical since the distribution of the interesting part of the data can not be decided statically. For an in-place rendering library design, implementing efficient dynamic load balancing strategy is challenging. Especially, in a runtime visualization setting, one or more simulation time steps are typically followed by one visualization step. The separation makes both the overlapping rendering scheme described previously less attractive and the design of a dynamic load balancing strategy a non-trivial task.

The rendering process is separated into local resampling and global compositing. Consequently, no communication is required during the resampling process. The parallel resampling algorithm is based on the author's previous work [4] which requires replicating voxels at boundaries. As most CFD codes use a finite-difference formulation, replication is required

anyway for a parallel implementation. Nevertheless, the replication requirement could be removed if special care were taken to implement resampling near the boundaries.

Another important issue to consider is the memory constraints of a processor as most applications are memory-intensive. It is usually not the case that the simulation and the renderer can share the data. Additional memory space is needed to store the data to be rendered, rendering parameters, and partial images. Normally, the quantities to be visualized are calculated and stored separately by the application, and passed to the renderer. Therefore, a PVR library should have moderate and predictable memory requirements so the application programmer can plan accordingly. The amount of memory space for the renderer is dominated by

$$\begin{aligned} &\textit{volume\_data\_size} + \textit{final\_sub\_image\_size} + \textit{partial\_image\_size} \\ &\qquad = (n_{voxel}/p) + 4{\cdot}((n/p) + (n/p^{2/3})) \end{aligned} \tag{1}$$

in *bytes*, where n_{voxel} is the total number of voxels, p the number of processors, n the number of pixels in the final image; each voxel can take from 1 to as many as 12 bytes dependent upon the desired data resolution and rendering algorithms used. The average size of a partial image can be calculated as follows. Assuming the block data distribution, the average *depth overalp* per pixel is $p^{1/3}$. Consequently, there are a total of $p^{1/3}n$ partial image pixels (ray segments), and the average number of pixels per processor is $(p^{1/3}n)/p = n/p^{2/3}$. Note that storage for boundary replication and rendering parameters are ignored.

After the rendering step, a partial image is produced on each processor. A parallel image compositing process then merges all partial images, in *depth order*, to achieve the complete image. This global combining process requires inter-processor communication. A direct compositing method [2, 7] has been selected for the library design, which proceeds independently of the way in which data was partitioned. In the direct compositing method, the image plane is subdivided and each node is assigned a subset of the total image pixels. Each rendered pixel is sent directly to the node assigned that portion of the image plane. Processing nodes accumulate these partial image pixels in an array and composite them in proper order after all rendering is completed. Neumann [7] subdivided the image space in an interleaved fashion to ensure load balancing. Other parallel image compositing algorithms such as binary-swap developed previously by the author [4], though superior in performance, would require regular data partitioning and the use of special data structures, and are thus less desirable for a generic PVR library design.

The scalability of the rendering phase is generally good since no communication is required. On the other hand, the scalability of the compositing part needs to be verified since there is some inter-processor communication involved. A performance analysis of the parallel compositing algorithm determines the communication cost for block data subdivision on a

bidirectional torus to be

$$c_1 \cdot p^{-(1/6)} \cdot n \cdot t_{trans} + c_2 \cdot p^{5/6} \cdot t_{overhead} \tag{2}$$

where c_1 and c_2 are positive constants smaller than 1, t_{trans} is per-pixel transfer cost, and $t_{overhead}$ is per-message overhead. Our analysis, consistent with Neumann's results [7], shows that the communication cost in fact decreases as the number of processors used increases. Thus the direct compositing method is acceptable.

At the end of image compositing, subimages are combined for storing or displaying. A dedicated HiPPI frame buffer and Parallel I/O support, when available, can be used for faster display rates and file I/O, respectively. In our setting, the host processor invokes a library function responsible for collecting and combining the subimages; the final image is displayed on a remote workstation. This results in a serial data stream which limites the display rates. The display rates are further degraded by network latency. Image compression techniques can be applied to alleviate these problems. In our current implementation, direct transfer is used because most large-scale scientific simulations do not run multiple time steps per second, even on a massively parallel computer.

Presently, the development of the library has been done on the 72-node Intel Paragon XP/S operated at the NASA Langley Research Center. The library has been written in C/C++ and the Intel's native communication library NX. Better portability can be achieved by moving to MPI or PVM. Finally, the library interface design includes both the API and the interactive user interface for setting up the renderer. The API should be as simple as possible. The application program passes the renderer the pointer to the subvolume, a record describing the geometry of the subvolume, and the rendering specifications. The interactive user interface, which has not been implemented, should allow the user to set and change the rendering specifications including viewing and lighting parameters, image resolution, rendering rate and transfer functions.

4 A Parallel Navier-Stokes Solver

A parallel Navier-Stokes solver has been implemented particularly for testing the PVR library on the Intel Paragon. It allows us to experiment with different data distribution schemes and other design criteria. The unsteady compressible Navier-Stokes equations are a mixed set of hyperbolic-parabolic equations in time. We chose the classical MacCormack finite-difference method [5] for the solution of the equations. This explicit, two-step, three-time level scheme is second-order accurate in both time and space. During each time level, the first step (predictor) calculates a temporary "predicted" value, and the second step (corrector) determines the final "corrected" value. Forward and backward differencing are alternated between the predictor and corrector steps to avoid any bias due to one-sided differencing and

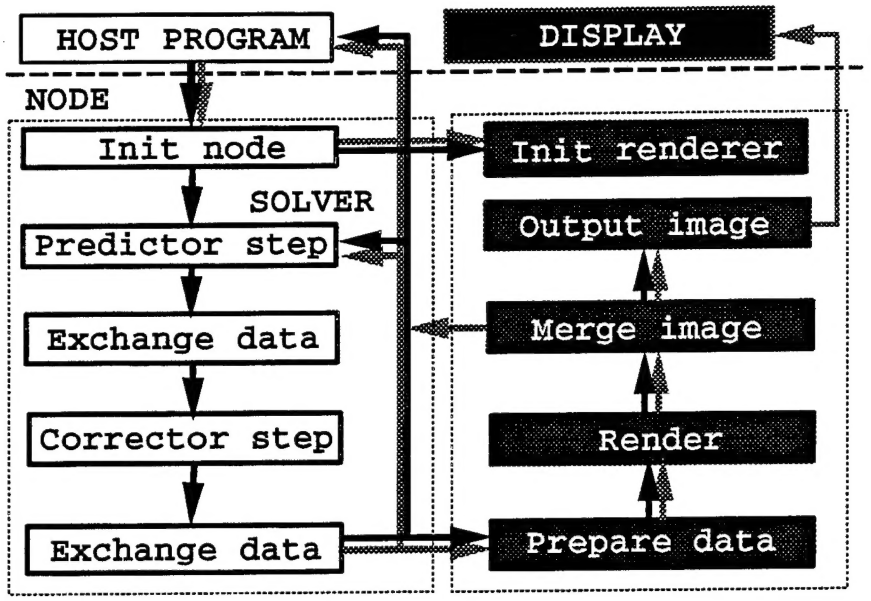


Figure 1: A setting for interactive monitoring.

to achieve second-order accuracy. The MacCormack method is used because it is easier to parallelize and implement. Later, for further testing of our rendering library, we will acquire more sophisticated CFD codes that are capable of simulating real-life problems.

The flow solver is implemented as a host-node model. As shown in Figure 1, the host program reads in the model specifications as well as the rendering specifications, and broadcasts them to every node processor. If the data domain is subdivided into blocks, each block has six neighbors. At each time level, the solution values at the outermost boundary layers of each block are exchanged between nearest neighbors after each predictor and corrector step. *"Prepare Data"* is a routine written by the application programmer to calculate the flow properties to be visualized and store in the appropriate format for rendering.

5 Test Results

For testing, the simulation models laminar flow entering a rectangular region. Flow at 100 meters per second enters a small square inlet at the left of the rectangular domain. The domain has a full width outlet at the right end, and is otherwise enclosed by walls. No body forces or external heat are assumed. Figure 2 presents direct volume visualization of vorticity values of the overall simulated domain at selected time steps. To see the flow structures appearing in these images, one must adjust either the color or the opacity mapping to enhance a particular range of values of interest. For example, in Figure 2, regions of high vorticity are enhanced by mapping high vorticity values to red and higher opacity values, and low vorticity values to white and lower opacity values. The symmetric structures

node size	8	27	64	125	216
simulation (compute)	10.665	3.1435	1.3282	0.7542	0.4109
simulation (communication)	0.0453	0.0441	0.029	0.0246	0.0199
preparing data	0.3026	0.0963	0.051	0.0670	0.0433
resampling	13.01	4.21	1.877	1.102	0.681
compositing (raw compositing)	0.151	0.0525	0.0501	0.0455	0.0310
compositing (communication)	0.106	0.080	0.0672	0.0612	0.0583

Table 1: Time Breakdown for Using Different Number of Nodes.

shown in all images verify the symmetrical boundary condition assigned. Three-dimensional volume visualization gives us a global, realistic view of the simulated flow, and thus a better understanding of the overall flow structures.

To evaluate the performance of both the simulation and the library, tests were performed on the Intel Paragon XP/S by using from eight to 216 compute nodes. The performance measures shown here were obtained by first calculating the average time over 300 time steps for three randomly picked viewing angles at each node; then the maximum time among the nodes is picked. Time is shown in seconds.

To see the scalability of both the solver and the rendering process, Table 1 shows time breakdown for rendering 256×256-pixel image on a fixed problem size of 60^3 data points using various numbers of processors. Figure 3 displays the same information graphically for revealing scalability by using log scale for both the x and y axes. As the timing results show, the simulation scales very well and thus achieves linear speedup. It also can be predicted that when running on a parallel system with faster processors and communication network, interactive steering of the simulation is feasible. Note that the times for image I/O are not shown here because our current setting for image data transfer and display could be optimized; thus the timing results obtained are not representative.

Table 2 shows the time breakdown of one time step of the flow simulation and rendering 256×256-pixel image using 125 nodes. Note that the *raw compositing* time is the average time that each processor takes to calculate the final pixel values for its responsible subimage area. The time a processor takes to collect all the needed ray segments from other processors is recorded in the *communication* time. To show how the rendering process perform as the problem size increases the tests were repeated for three different domain sizes: 60^3, 120^3 and 240^3. It is easy to see that simulation time dominates for typical problem sizes. From both Table 1 and 2, we can also conclude that the rendering process scales well with the simulation as well as the parallel system. However, the compositing times, though declining, do not scale as good as the resampling times due to the increasing numbers of processors used.

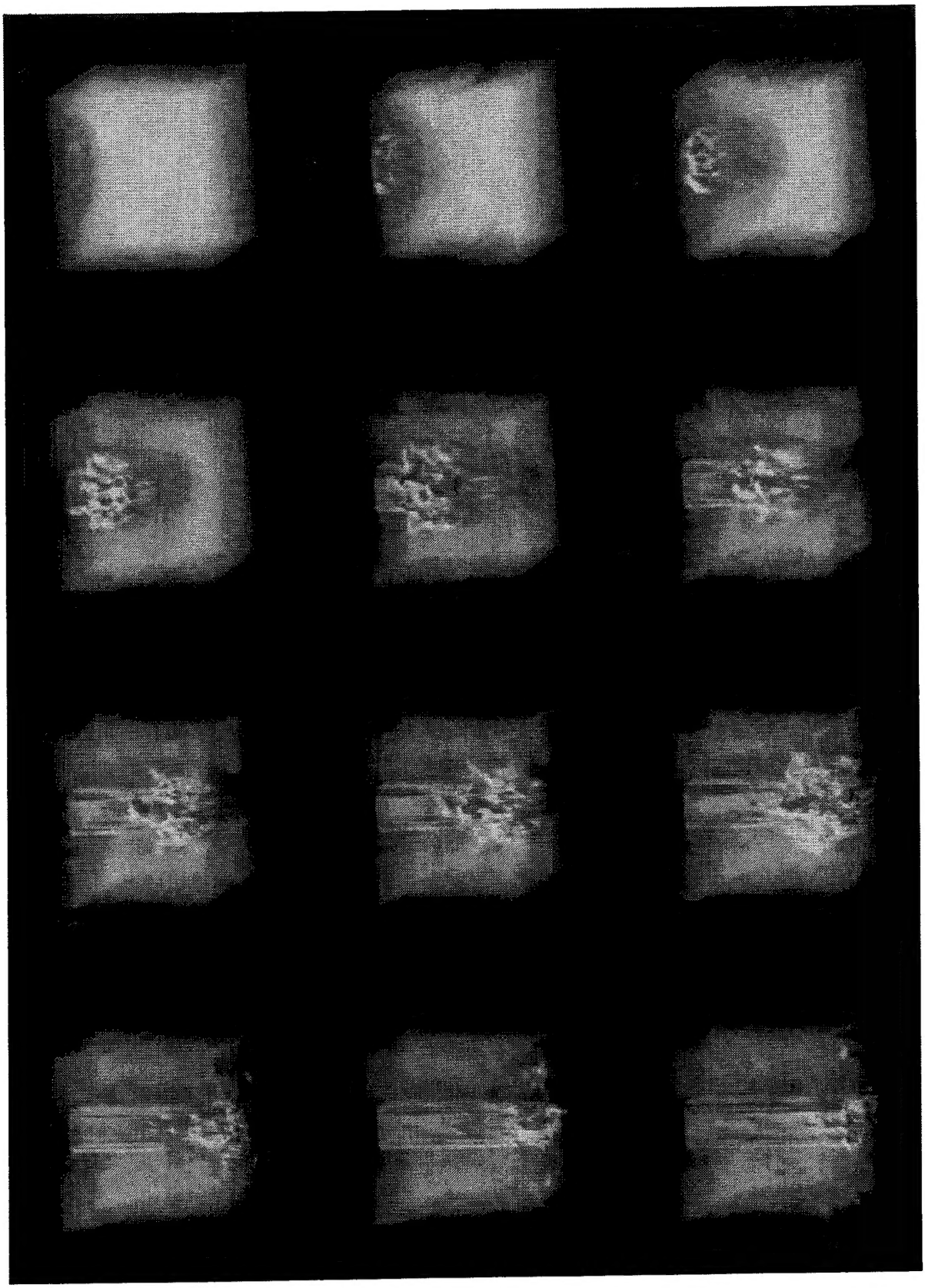


Figure 2. Tracking the Magnitude of Vorticity at Selected Time Steps.

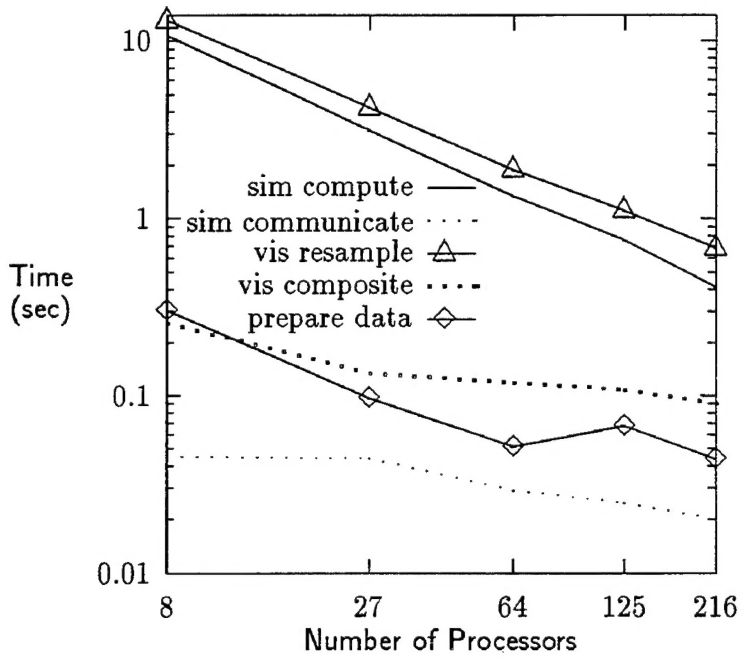


Figure 3: Timing for Both the Simulation and Runtime Visualization.

problem size	60^3	120^3	240^3
simulation (compute)	0.7542	5.6258	44.0
simulation (communication)	0.0246	0.073	0.254
preparing data	0.067	0.261	1.485
resampling	1.102	2.054	2.845
compositing (raw compositing)	0.0455	0.044	0.044
compositing (communication)	0.0612	0.085	0.0765

Table 2: Time Breakdown for Different Problem Sizes.

6 Conclusions

If the current trend in scientific computing continues, a parallel PVR library could be very useful to computational researchers who run their simulations on massively parallel computers. Interactive visual responses reflecting simulation states and the physical phenomena modeled allow better control of the simulation, and can offer additional insights into the physics behind the model. The parallel rendering library described in this paper is designed for distributed memory message passing environments. It provides an interface to handle volume data on a rectilinear grid. In CFD, rectilinear grids are still widely used but a majority of problems of more complex geometry require the use of either curvilinear or unstructured grids. This library can be extended to handle those grids using the algorithm described in [3]. Important future work includes implementing dynamic load balancing strategies, a graphical user interface, and improving image I/O.

References

[1] CROCKETT, T. Design Considerations For Parallel Graphics Libraries. Tech. rep., Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681-0001, June 1994. ICASE Report No. 94-49.

[2] HSU, W. M. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings 1993 Parallel Rendering Symposium* (October 1993), ACM SIGGRAPH, pp. 7–14.

[3] MA, K.-L. Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *Proceedings 1995 Parallel Rendering Symposium* (October 1995), ACM SIGGRAPH.

[4] MA, K.-L., PAINTER, J., HANSEN, C., AND KROGH, M. Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68.

[5] MACCORMACK, R. The Effect of Viscosity in Hypervelocity Impact Cratering. American Institute of Aeronautics and Astronautics. AIAA Paper No. 69-354.

[6] MACKERRAS, P., AND CORRIE, B. Exploiting Data Coherence to Improve Parallel Volume Rendering. *IEEE Parallel & Distributed Technology 2*, 2 (1994), 8–16.

[7] NEUMANN, U. Parallel Volume-Rendering Algorithm performance on Mesh-Connected Multicomputers. In *Proceedings 1993 Parallel Rendering Symposium* (October 1993), ACM SIGGRAPH, pp. 97–104.

[8] NIEH, J., AND LEVOY, M. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *1992 Workshop on Volume Visualization* (1992), pp. 17–24. Boston, October 19-20.

[9] SILVA, C. T., AND KAUFMAN, A. E. Parallel Performance Measures for Volume Ray Casting. In *Proceedings Visualization '94* (1994), pp. 196–203.

[10] ZHU, J. On the Implementation Issues of Domain Decomposition Algorithms for Parallel Computers. In *Proceedings Parallel CFD '92* (1992), pp. 427–438.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
October 1995

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
RUNTIME VOLUME VISUALIZATION FOR PARALLEL CFD

5. FUNDING NUMBERS
C NAS1-19480
WU 505-90-52-01

6. AUTHOR(S)
Kwan-Liu Ma

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 95-74

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-198232
ICASE Report No. 95-74

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Dennis M. Bushnell
Final Report
To appear in the Proceedings of the Parallel CFD '95 Conference

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 60, 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper discusses some aspects of the design of a data distributed, massively parallel volume rendering library for runtime visualization of parallel computational fluid dynamics simulations in a message-passing environment. Unlike the traditional scheme in which visualization is a postprocessing step, the rendering is done in place on each node processor. Computational scientists who run large-scale simulations on a massively parallel computer can thus perform interactive monitoring of their simulations. The current library provides an interface to handle volume data on rectilinear grids. The same design principles can be generalized to handle other types of grids. For demonstration, we run a parallel Navier-Stokes solver making use of this rendering library on the Intel Paragon XP/S. The interactive visual response achieved is found to be very useful. Performance studies show that the parallel rendering process is scalable with the size of the simulation as well as with the parallel computer.

14. SUBJECT TERMS
Volume Rendering; Parallel Processing; Computational Fluid Dynamics

15. NUMBER OF PAGES
13

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		